Memory Hierarchy (II)

Hung-Wei Tseng
Recap: von Neumann architecture

Processor

PC

memory

120007a30: 0f00bb27  ldah gp,15(t12)
120007a34: 509cbd23  lda gp,-25520(gp)
120007a38: 00005d24  ldah t1,0(gp)
120007a3c: 0000bd24  ldah t4,0(gp)
120007a40: 2ca422a0  ldl t0,-23508(t1)
120007a44: 130020e4  beq t0,120007a94
120007a48: 00003d24  ldah t0,0(gp)
120007a4c: 2ca4e2b3  stl zero,-23508(t1)
120007a50: 0004ff47  clr v0
120007a54: 28a4e5b3  stl zero,-23512(t4)
120007a58: 20a421a4  ldq t0,-23520(t0)
120007a5c: 0e0020e4  beq t0,120007a98
120007a60: 0204e147  mov t0,t1
120007a64: 0304ff47  clr t2
120007a68: 0500e0c3  br 120007a80
Recap: The memory gap problem

The access time of DRAM is around 50ns, 100x to the cycle time of a 2GHz processor!

SRAM is as fast as the processor, but

```assembly
lw   $t2, 0($a0)
add  $t3, $t2, $a1
addi $a0, $a0, 4
subi $a1, $a1, 1
bne  $a1, LOOP
lw   $t2, 0($a0)
add  $t3, $t2, $a1
```

The access time of DRAM is around 50ns, 100x to the cycle time of a 2GHz processor!

SRAM is as fast as the processor, but

---

**Memory technology** | **Typical access time** | **$ per GiB in 2012**
---|---|---
SRAM semiconductor memory | 0.5–2.5 ns | $500–$1000
DRAM semiconductor memory | 50–70 ns | $10–$20
Flash semiconductor memory | 5,000–50,000 ns | $0.75–$1.00
Magnetic disk | 5,000,000–20,000,000 ns | $0.05–$0.10
Memory hierarchy
The memory hierarchy

- Fastest, Most Expensive
  - CPU
    - 32* 64-bit registers
    - L1: 16KB-64KB
    - L2: 128KB-512KB
    - L3: Several MBs
- Access time
  - < 1ns
  - < 1ns ~ 20 ns
  - 100ns
  - 10,000,000ns
- Biggest
  - Several GBs
  - 500+ GB
  - Secondary Storage
  - Main Memory
  - Cache
Recap: Localities in your code

- Spatial locality: programs tend to access neighboring data/instructions
  - Data structures (e.g. arrays) demonstrate strong spatial locality
  - Especially effective for code/instructions — you usually just move to the next instruction or loop back to the small piece of code

- Temporal locality: programs tend to have frequently accessed data
  - You may update/reference the same set of memory locations many times in your code
Recap: Architecting caches to capture localities

• To capture spatial locality
  • We need to put not only just a “word” or small piece of data/instructions, but a “block” of data/instructions
  • A tag associated with each block

• To capture temporal locality
  • A cache replacement policy to keep most frequently used data (e.g. LRU)
  • LRU — kick out the least recently used block when we need to kick out one

• Performance needs to be better than linear search
  • Make cache a hardware hash table!
  • The hash function takes memory addresses as inputs
**The structure of a cache**

**Set:** cache blocks/lines sharing the same index. A cache is called N-way set associative cache if N blocks share the same set/index (this one is a 2-way set cache)

**Tag:**
- the high order address bits stored along with the data in a block to identify the actual address of the cache line.

**Block / Cacheline:** The basic unit of data storage in cache. Contains all data with the same tag/prefix and index in their memory addresses.

**Valid:** if the data is meaningful

**Dirty:** if the block is modified

<table>
<thead>
<tr>
<th>valid</th>
<th>dirty</th>
<th>tag</th>
<th>data</th>
<th>valid</th>
<th>dirty</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1 0 1000 0001 0000 1000 0000</td>
<td></td>
<td></td>
<td></td>
<td>1 1 1000 0000 0000 0000 0000 0000</td>
<td></td>
</tr>
</tbody>
</table>
Accessing the cache

Hit: The data was found in the cache
Miss: The data was not found in the cache

Offset: The position of the requesting word in a cache block
Outline

• Cache organization (cont.)
• How does cache interact with the processor
• Performance evaluation with cache
• Optimizing cache performance and your code!
How many bits in each field?

- **lg(number of sets)**
- **lg(block size)**

```
tag
index
offset
valid
dirty
tag
data
valid
dirty
tag
data
hit?
hit?
```
C = ABS

- **C**: Capacity in data arrays
- **A**: Way-Associativity
  - N-way: N blocks in a set, A = N
  - 1 for direct-mapped cache
- **B**: Block Size (Cacheline)
  - How many bytes in a block
- **S**: Number of Sets:
  - A set contains blocks sharing the same index
  - 1 for fully associate cache
Corollary of $C = \text{ABS}$

- offset bits: $\lg(B)$
- index bits: $\lg(S)$
- tag bits: $\text{address}_{\text{length}} - \lg(S) - \lg(B)$
  - $\text{address}_{\text{length}}$ is 32 bits for 32-bit machine
- $(\text{address} / \text{block}_{\text{size}}) \% S = \text{set index}$
L1 data (D-L1) cache configuration of AMD Phenom II
- Size 64KB, 2-way set associativity, 64B block
- Assume 64-bit memory address

Which of the following is correct?

A. Tag is 49 bits
B. Index is 8 bits
C. Offset is 7 bits
D. The cache has 1024 sets
E. None of the above

\[ C = \text{ABS} \]
\[ 64KB = 2 \times 64 \times S \]
\[ S = 512 \]
\[ \text{offset} = \lg(64) = 6 \text{ bits} \]
\[ \text{index} = \lg(512) = 9 \text{ bits} \]
\[ \text{tag} = 64 - \lg(512) - \lg(64) = 49 \text{ bits} \]
• L1 data (D-L1) cache configuration of Core i7
  • Size 32KB, 8-way set associativity, 64B block
  • Assume 64-bit memory address
  • Which of the following is NOT correct?
    A. Tag is 52 bits
    B. Index is 6 bits
    C. Offset is 6 bits
    D. The cache has 128 sets

\[
C = \text{ABS} \\
32\text{KB} = 8 \times 64 \times S \\
S = 64 \\
\text{offset} = \lg(64) = 6 \text{ bits} \\
\text{index} = \lg(64) = 6 \text{ bits} \\
\text{tag} = 64 - \lg(64) - \lg(64) = 52 \text{ bits}
\]
Put everything all together:
How cache interacts with CPU
What happens on a read?

- Read hit
  - hit time
- Read miss?
  - Select victim block
    - LRU, random, FIFO, ...
    - Write back if dirty — will talk later
  - Fetch Data from Lower Memory Hierarchy
    - As a unit of a cache block
      - Data with the same “block address” will be fetch
    - Miss penalty
Special case: a direct-mapped cache

Tag: the high order address bits stored along with the data to identify the actual address of the cache line.

Hit: The data was found in the cache
Miss: The data was not found in the cache

Block (cacheline): The basic unit of data storage in cache. Contains all data with the same tag and index in their address

memory address: 1000 0000 0000 0000 0000 0001 0101 1000

Tag:

data

valid

dirty

hit? miss?

1000 0000 0000 0000 0000
Simulate a direct-mapped cache

- Consider a direct mapped (1-way) cache with 16 blocks, a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000

- \( C = A B S \)
- \( S = \frac{256}{(16 \times 1)} = 16 \)
- \( \lg(16) = 4 \): 4 bits are used for the index
- \( \lg(16) = 4 \): 4 bits are used for the byte offset
- The tag is 48 - (4 + 4) = 40 bits
- For example: 0b1000 0000 0000 0000 0000 0000 1000 0000
Simulate a direct-mapped cache

<table>
<thead>
<tr>
<th>valid</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0b10</td>
<td></td>
</tr>
</tbody>
</table>

- 0b10 0000 0000  miss
- 0b10 0000 1000  hit!
- 0b10 0001 0000  miss
- 0b10 0001 0100  hit!
- 0b11 0001 0000  miss
- 0b10 0000 0000  hit!
- 0b10 0000 1000  hit!
- 0b10 0001 0000  miss
- 0b10 0001 0100  hit!
Simulate a 2-way cache

- Consider a 2-way cache with 16 blocks (8 sets), a block size of 16 bytes, and the application repeatedly reading the following memory addresses:
  - 0b1000000000, 0b1000001000, 0b1000010000, 0b1000010100, 0b1100010000

- $8 = 2^3$: 3 bits are used for the index
- $16 = 2^4$: 4 bits are used for the byte offset
- The tag is $32 - (3 + 4) = 25$ bits
- For example: 0b1000 0000 0000 0000 0000 0000 0000 0001 0000
Simulate a 2-way cache

<table>
<thead>
<tr>
<th>v</th>
<th>tag</th>
<th>data</th>
<th>v</th>
<th>tag</th>
<th>data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1 0b100</td>
<td></td>
<td>1</td>
<td>0b100</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1 0b100</td>
<td>1 0b110</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4 5 6 7</td>
<td></td>
<td>8</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
0b10 0000 0000  miss
0b10 0000 1000  hit!
0b10 0000 1000  miss
0b10 0001 0000  hit!
0b10 0001 0000  hit!
0b11 0001 0000  miss
0b10 0000 0000  hit!
0b10 0000 1000  hit!
0b10 0001 0000  hit!
0b10 0001 0100  hit!
```

### tag index

```
0b10 0000 0000  miss
0b10 0000 1000  hit!
0b10 0001 0000  miss
0b10 0001 0100  hit!
```

22
• D-L1 Cache configuration of AMD Phenom II
  • Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and
    assuming 32-bit address.
  • Cache performance for the following code?
    ```
    int a[16384], b[16384], c[16384];
    /* c = 0x10000, a = 0x20000, b = 0x30000 */
    for(i = 0; i < 512; i++) {
        c[i] = a[i] + b[i];
        // load a, b, and then store to c
    }
    ```
  • What’s the data cache miss rate for this code?
    A. 6.25%
    B. 56.25%
    C. 66.67%
    D. 68.75%
    E. 100%

\[
\begin{align*}
\text{C} & = \text{ABS} \\
\text{64KB} & = 2 \times 64 \times S \\
\text{S} & = 512 \\
\text{offset} & = \log(64) = 6 \text{ bits} \\
\text{index} & = \log(512) = 9 \text{ bits} \\
\text{tag} & = 64 - \log(512) - \log(64) = 49 \text{ bits}
\end{align*}
\]
AMD Phenom II

- Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 48-bit address.

```c
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
```

<table>
<thead>
<tr>
<th>address in hex</th>
<th>tag</th>
<th>address in binary</th>
<th>index</th>
<th>offset</th>
<th>tag</th>
<th>index</th>
<th>hit? miss?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a[0]</td>
<td>0x20000</td>
<td>10 0000 0000 0000 0000</td>
<td>0x4</td>
<td>0</td>
<td>miss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>11 0000 0000 0000 0000</td>
<td>0x6</td>
<td>0</td>
<td>miss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>1 0000 0000 0000 0000</td>
<td>0x2</td>
<td>0</td>
<td>miss, evict 0x4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>10 0000 0000 0000 0100</td>
<td>0x4</td>
<td>0</td>
<td>miss, evict 0x6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>11 0000 0000 0000 0100</td>
<td>0x6</td>
<td>0</td>
<td>miss, evict 0x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>1 0000 0000 0000 0100</td>
<td>0x2</td>
<td>0</td>
<td>miss, evict 0x4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load a[15]</td>
<td>0x2003C</td>
<td>10 0000 0000 0011 1100</td>
<td>0x4</td>
<td>0</td>
<td>miss, evict 0x6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load b[15]</td>
<td>0x3003C</td>
<td>11 0000 0000 0011 1100</td>
<td>0x6</td>
<td>0</td>
<td>miss, evict 0x2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>store c[15]</td>
<td>0x1003C</td>
<td>1 0000 0000 0011 1100</td>
<td>0x2</td>
<td>0</td>
<td>miss, evict 0x4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load a[16]</td>
<td>0x20040</td>
<td>10 0000 0000 0100 0000</td>
<td>0x4</td>
<td>1</td>
<td>miss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>load b[16]</td>
<td>0x30040</td>
<td>11 0000 0000 0100 0000</td>
<td>0x6</td>
<td>1</td>
<td>miss</td>
<td></td>
<td></td>
</tr>
<tr>
<td>store c[16]</td>
<td>0x10040</td>
<td>1 0000 0000 0100 0000</td>
<td>0x2</td>
<td>1</td>
<td>miss, evict 0x4</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

100% miss rate!
• D-L1 Cache configuration of AMD Phenom II
  • Size 64KB, 2-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 32-bit address.
  • Cache performance for the following code?
    • int a[16384], b[16384], c[16384];
      /* c = 0x10000, a = 0x20000, b = 0x30000 */
      for (i = 0; i < 512; i++) {
        c[i] = a[i] + b[i];
        //load a, b, and then store to c
      }
    • What’s the data cache miss rate for this code?
      A. 6.25%
      B. 56.25%
      C. 66.67%
      D. 68.75%
      E. 100%
• D-L1 Cache configuration of intel Core i7 processor
  • Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.
  • Cache performance for the following code?
    • int a[16384], b[16384], c[16384];
      /* c = 0x10000, a = 0x20000, b = 0x30000 */
      for(i = 0; i < 512; i++) {
        c[i] = a[i] + b[i];
        //load a, b, and then store to c
      }
    • What’s the data cache miss rate for this code?
      A. 6.25%
      B. 56.25%
      C. 66.67%
      D. 68.75%
      E. 100%
int a[16384], b[16384], c[16384];
/* c = 0x10000, a = 0x20000, b = 0x30000 */
for(i = 0; i < 512; i++)
{
    c[i] = a[i] + b[i]; /*load a[i], load b[i], store c[i]*/
}

<table>
<thead>
<tr>
<th>address</th>
<th>tag</th>
<th>index</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>load a[0]</td>
<td>0x20000</td>
<td>0x20</td>
<td>0</td>
</tr>
<tr>
<td>load b[0]</td>
<td>0x30000</td>
<td>0x30</td>
<td>0</td>
</tr>
<tr>
<td>store c[0]</td>
<td>0x10000</td>
<td>0x10</td>
<td>0</td>
</tr>
<tr>
<td>load a[1]</td>
<td>0x20004</td>
<td>0x20</td>
<td>0</td>
</tr>
<tr>
<td>load b[1]</td>
<td>0x30004</td>
<td>0x30</td>
<td>0</td>
</tr>
<tr>
<td>store c[1]</td>
<td>0x10004</td>
<td>0x10</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>load a[15]</td>
<td>0x2003C</td>
<td>0x20</td>
<td>0</td>
</tr>
<tr>
<td>load b[15]</td>
<td>0x3003C</td>
<td>0x30</td>
<td>0</td>
</tr>
<tr>
<td>store c[15]</td>
<td>0x1003C</td>
<td>0x10</td>
<td>0</td>
</tr>
<tr>
<td>load a[16]</td>
<td>0x20040</td>
<td>0x20</td>
<td>1</td>
</tr>
<tr>
<td>load b[16]</td>
<td>0x30040</td>
<td>0x30</td>
<td>1</td>
</tr>
<tr>
<td>store c[16]</td>
<td>0x1003C</td>
<td>0x10</td>
<td>1</td>
</tr>
</tbody>
</table>

\[
32 \times 3 / (512 \times 3) = 1/16 = 6.25\% \text{ (93.75\% hit rate!)}
\]
• D-L1 Cache configuration of intel Core i7 processor
  • Size 32KB, 8-way set associativity, 64B block, LRU policy, write-allocate, write-back, and assuming 64-bit address.
  • Cache performance for the following code?
    * int a[16384], b[16384], c[16384];
      /* c = 0x10000, a = 0x20000, b = 0x30000 */
      for(i = 0; i < 512; i++) {
        c[i] = a[i] + b[i];
        //load a, b, and then store to c
      }
    • What’s the data cache miss rate for this code?
      A. 6.25%
      B. 56.25%
      C. 66.67%
      D. 68.75%
      E. 100%

C = ABS
32KB = 8 * 64 * S
S = 64
offset = lg(64) = 6 bits
index = lg(64) = 6 bits
tag = 64 - lg(64) - lg(64) = 52 bits
Way associativity and cache performance

![Graph showing miss rate for different cache sizes and associativities.](image)
Pros & cons of way-associate caches

• Help alleviating the hash collision by having more blocks associating with each different index.
  • N-way associative: the block can be in N blocks of the cache
• Fully associative
  • The requested block can be anywhere in the cache
  • Or say N = the total number of cache blocks in the cache
• Slower
  • Increasing associativity requires multiple tag checks
  • N-Way associativity requires N parallel comparators
  • This is expensive in hardware and potentially slow.
  • This limits associativity L1 caches to 2-8.
  • Larger, slower caches can be more associative
What happens on a write? (Write Allocate, write back)

- Write hit?
  - Update in-place
  - Set dirty bit (Write-Back Policy)

- Write miss?
  - Select victim block
    - LRU, random, FIFO, ...
    - Write back to lower memory hierarchy if dirty
  - Fetch Data from Lower Memory Hierarchy
    - As a unit of a cache block
    - Miss penalty
Performance evaluation considering cache
Multi-layer caches

- Speed of L1 matches the processor
- Caches data/code as many as possible in L2/L3 to avoid DRAM accesses

![Diagram showing multi-layer caches with CPU, L1, L2, L3, Main Memory, and Secondary Storage]
Performance evaluation considering cache

- If the load/store instruction hits in L1 cache where the hit time is usually the same as a CPU cycle
  - The CPI of this instruction is the base CPI
- If the load/store instruction misses in L1, we need to access L2
  - The CPI of this instruction needs to include the cycles of accessing L2
- If the load/store instruction misses in both L1 and L2, we need to go to lower memory hierarchy (L3 or DRAM)
  - The CPI of this instruction needs to include the cycles of accessing L2, L3, DRAM
How to evaluate cache performance

- CPI\text{Average} : the average CPI of a memory instruction

\[
\text{CPI}_{\text{Average}} = \text{CPI}_{\text{base}} + \text{miss\_rate}_{L1} \times \text{miss\_penalty}_{L1}
\]

\[
\text{miss\_penalty}_{L1} = \text{CPI}_{\text{accessing\_L2}} + \text{miss\_rate}_{L2} \times \text{miss\_penalty}_{L2}
\]

\[
\text{miss\_penalty}_{L2} = \text{CPI}_{\text{accessing\_L3}} + \text{miss\_rate}_{L3} \times \text{miss\_penalty}_{L3}
\]

\[
\text{miss\_penalty}_{L3} = \text{CPI}_{\text{accessing\_DRAM}} + \text{miss\_rate}_{DRAM} \times \text{miss\_penalty}_{DRAM}
\]

- If the problem is asking for **average memory access time**, transform the CPI values into/from time by multiplying with CPU cycle time!
Average memory access time

- Average Memory Access Time (AMAT) = Hit Time + Miss rate * Miss penalty
  - Miss penalty = AMAT of the lower memory hierarchy
  - AMAT = hit\_time\textsubscript{L1} + miss\_rate\textsubscript{L1} * AMAT\textsubscript{L2}
    - AMAT\textsubscript{L2} = hit\_time\textsubscript{L2} + miss\_rate\textsubscript{L2} * AMAT\textsubscript{DRAM}
Cache & Performance

- 5-stage MIPS processor.
  - Application: 80% ALU, 20% Loads
  - L1 I-cache miss rate: 5%, hit time: 1 cycle
  - L1 D-cache miss rate: 10%, hit time: 1 cycle
  - L2 U-Cache miss rate: 20%, hit time: 10 cycles
  - Main memory hit time: 100 cycles
  - Assume the program is read only (nothing dirty)
  - What’s the average CPI?

A. 1.1
B. 1.6
C. 2.1
D. 3.1
E. none of the above

\[
\text{CPI}_{\text{Average}} = \text{CPI}_{\text{base}} + \text{miss rate} \times \text{miss penalty}
\]

\[
= 1 + 100\% \times (5\% \times (10 + 20\% \times (1 \times 100))) \\
+ 20\% \times (10\% \times (10 + 20\% \times (1 \times 100)))
\]

\[
= 3.1
\]
Cache & Performance

- Application: 80% ALU, 20% Loads
- L1 I-cache miss rate: 5%, hit time: 1 cycle
- L1 D-cache miss rate: 10%, hit time: 1 cycle
- L2 U-Cache miss rate: 20%, hit time: 10 cycles
- Main memory hit time: 100 cycles
- What’s the average CPI?

Average CPI = CPI_base + miss_rate * miss_penalty

= 1 + 100% * (5% * (10 + 20% * (1 * 100)))
+ 20% * (10% * (1) * (10 + 20% * ((1) * 100)))

= 3.1